

Java CM3

Olivier Marchetti

Laboratoire d'informatique de Paris 6 – Pôle SoC – Sorbonne Université

9 novembre 2021



Plan

- 1 Exceptions
 - Intérêt
 - Bloc try/catch
 - Bloc finally
- 2 Classe Abstraite – Interface
 - Classe Abstraite
 - Interface
- 3 Duplication d'objets
 - Surface/Profondeur
 - Clonage d'objet
- 4 Quelques approfondissements
 - La classe Object et ses méthodes
 - Introspection en JAVA
 - Classes génériques
 - Classes enveloppes
 - Collections
 - Types énumérés



1 Exceptions

- Intérêt
- Bloc try/catch
- Bloc finally

2 Classe Abstraite – Interface

- Classe Abstraite
- Interface

3 Duplication d'objets

- Surface/Profondeur
- Clonage d'objet

4 Quelques approfondissements

- La classe Object et ses méthodes
- Introspection en JAVA
- Classes génériques
- Classes enveloppes
- Collections
- Types énumérés

Le traitement des exceptions

Une force de JAVA provient de sa sûreté en traitant dynamiquement les cas d'erreur :

- ❶ Erreur : problème grave à l'exécution \Rightarrow **épuisement mémoire**
- ❷ Exception : problème grave lié à la conception \Rightarrow **division par zéro**

Une exception correspond à une situation non-prévue :

```
Si (situation est « TypeSituation ») alors  
    « lancer » exception de type TypeSituation
```

- ▶ Prévoir une classe TypeSituation;
- ▶ L'exception TypeSituation devra étendre java.lang.Exception.

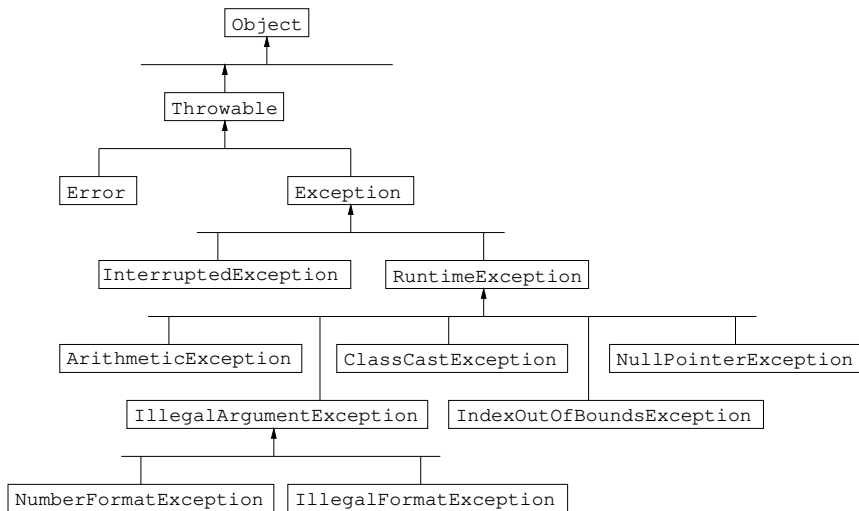
À l'exécution, la JVM :

- ❶ détectera ces exceptions;
- ❷ déroutera provisoirement le flot d'instructions;
- ❸ recouvrera le flot d'instructions.

```
★ ★ ★ ★ ★ ★ ★ ★ ★  
★  Traitement efficace !  ★  
★ ★ ★ ★ ★ ★ ★ ★ ★
```

Le traitement des exceptions - arborescence

L'API JAVA prévoit de très nombreuses exceptions :



Le traitement des exceptions – syntaxe

Pour traiter une instruction susceptible de « lever » une exception, on l'insère dans un bloc try-catch.

```
try {  
    instructions;  
} catch (TypeException1 e) {  
    instructionsAppropriées;  
} ...  
} catch (TypeExceptionN e) {  
    instructionsAppropriées;  
}  
... // Reprise du flot d'instructions.
```

- 1 La JVM scrute les éventuelles exceptions levées dans le bloc try;
- 2 Si une exception a été levée et qu'il existe un bloc catch adapté au type de cette exception, alors les instructions de ce bloc seront exécutées.
- 3 Reprise du code après le dernier bloc catch.

Le traitement des exceptions – mécanismes de propagation

Si (situation est « TypeSituation ») alors
« lancer » exception de type TypeSituation

Plusieurs possibilités pour le traitement des exceptions :

- ❶ Interceptor une exception et la traiter (avec bloc try-catch);
- ❷ Interceptor une exception, la traiter et la relancer avec throw;
- ❸ Laisser l'exception se propager à la méthode appelante, avec throws;

```
try {
    instructions
} catch (ExceptionType1 e) {
    instructionsAppropriées;
    throw e;
}
...
```

```
methode() throws TypeException1,
    ...,
    TypeExceptionN {
    instructionsMethode;
}
```

throws figure dans la signature.

- ❹ Laisser l'exception se propager jusqu'à la fin du programme.

```
class ExceptionTableau {
    public static void main(String args[]) {
        int tab[] = {1, 2, 3, 4};
        tab[4] = 5;
    }
}
```

```
[13:31][Prog pc666 :]$ java ExceptionTableau
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at ExceptionTableau.main(ExceptionTableau.java:4)
```

Informations utiles sur la pile des appels !

Le traitement des exceptions : exemple

La méthode `parseInt()`¹ forme un entier `x` à partir d'un objet `String` "`x`" :

- ▶ si "`x`" désigne bien l'écriture d'un entier en base 10 ;
- ▶ sinon génère/lève une exception de type `NumberFormatException`.

```
class StatLigneDeCommandes {
    static float moyenne(String argumentsEntier[]) {
        int somme = 0;
        int entierLu;
        int nbNotes = 0;
        for (String s : argumentsEntier) {
            try {
                entierLu = Integer.parseInt(s);
                somme += entierLu;
                nbNotes++;
            } catch (NumberFormatException e) {
                System.out.println("Argument " + s + " incorrect.");
            }
        }
        return ((float) somme) / nbNotes;
    }

    public static void main(String args[]) {
        System.out.println("La moyenne de votre série vaut " + moyenne(args));
    }
}
```

```
[15:03][Prog pc666 :]$ java StatLigneDeCommandes 11 17 4 9 13 14
La moyenne de votre série vaut 11.333333
[15:03][Prog pc666 :]$ java StatLigneDeCommandes 10 17 Rat-TaupéNu 12
Argument Rat-TaupéNu incorrect.
La moyenne de votre série vaut 13.0
```

¹<https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html>

Le traitement des exceptions : définir ses propres exceptions

L'héritage permet de définir ses propres exceptions.

- ▶ Exemple : gestion du cas où aucun argument n'est l'écriture d'un entier.

```
class MonException extends Exception {  
    int nbEntiersATraiter;  
  
    MonException(int nbEntiersATraiter) {  
        this.nbEntierATraiter = nbEntierATraiter;  
    }  
  
    public String toString() {  
        return "MonException : aucunes des " + nbEntierATraiter + " entrées n'est correcte."  
    }  
}
```

- ▶ Générer/lancer son exception :

```
e = new MonException(argumentsEntier.length);
```

- ▶ Propager l'exception après sa création :

```
throw e;
```

- ▶ Attraper son exception dans un bloc catch adapté :

```
catch (MonException e) {...}
```

Le traitement des exceptions : exemple – suite et fin

Si aucun argument saisi n'est l'écriture d'un entier :

```
[15:36][Prog pc666 :]$ java StatLigneDeCommandes Rat-TaupéNu Pyramides 3.1415
...
La moyenne de votre série vaut NaN
```

Dérouter le code avec throw/throws afin d'éviter cette division :

```
class StatLigneDeCommandes {
    static float moyenne(String argumentsEntier[]) throws MonException {
        ...
        for (String s : argumentsEntier) {
            try { ...
            } catch (NumberFormatException e) {
                ...
            }
        }
        if (nbNotes == 0) throw new MonException(argumentsEntier.length);
        return ((float) somme)/nbNotes;
    }

    public static void main(String args[]) {
        // Le traitement de l'exception se fait au niveau de la méthode appelante.
        try {
            System.out.println("La moyenne de votre serie vaut " + moyenne(args));
        } catch (MonException e) {
            System.out.println(e);
        }
    }
}
```



Le traitement des exceptions - les blocs finally

Pour exécuter des instructions quoi qu'il arrive dans le bloc try, on utilise un bloc finally :

```
try {  
    instructions;  
} catch (TypeException1 e) {  
    instructionsAppropriées;  
} ...  
} catch (TypeExceptionN e) {  
    instructionsAppropriées;  
} finally {  
    instructionsAppropriées;  
}  
... // Reprise du flot d'instructions.
```

javac : les exceptions non vérifiées & la clause throws

Certaines exceptions sont si courantes que javac dispense de spécifier des clauses throws : RuntimeException.

1 Exceptions

- Intérêt
- Bloc try/catch
- Bloc finally

2 Classe Abstraite – Interface

- Classe Abstraite
- Interface

3 Duplication d'objets

- Surface/Profondeur
- Clonage d'objet

4 Quelques approfondissements

- La classe Object et ses méthodes
- Introspection en JAVA
- Classes génériques
- Classes enveloppes
- Collections
- Types énumérés

Classe abstraite – définitions 1/4

Définition : une méthode abstraite est une méthode pour laquelle seule une signature est donnée.

```
modificateurAcces abstract type nomMethode([type1 arg1,..., typeN argN]);
```

- ▶ Précédée du mot clé **abstract**.
- ▶ Se termine par un point-virgule.

Définition : une classe sera dit abstraite dès lors que l'une de ses méthodes est abstraite.

```
modificateurAcces abstract class NomClasse {  
    champ1;  
    champ2;  
    ...  
    abstract methode();  
    ...  
}
```

Remarque

- ▶ Un constructeur ne peut être déclaré avec le mot clé **abstract**.
- ▶ Une classe abstraite définit un type.

Classe abstraite – usage 2/4

Attention : une classe abstraite ne peut **jamais** être instanciée.

```
abstract class ClasseAbstraite {  
    int champ;  
    abstract void methodeAbstraite();  
  
    ClasseAbstraite(int c) {  
        champ = c;  
    }  
  
    public static void main(String args[]) {  
        ClasseAbstraite objAbs = new ClasseAbstraite(1);  
    }  
}
```

Le compilateur javac veille et empêche toute instantiation !

```
[13:54][Prog pc666 :]$ javac ClasseAbstraite.java  
ClasseAbstraite.java:8: error: ClasseAbstraite is abstract; cannot be instantiated  
    ClasseAbstraite objAbs = new ClasseAbstraite(1);  
                                ^  
1 error
```


Classe abstraite – usage 3/4

Pour « utiliser » une classe abstraite, le programmeur doit d'abord l'étendre et définir les méthodes abstraites au sein des classes dérivées.

```
abstract class ClasseAbstraite {
    int champ;

    abstract void methodeAbstraite();

    ClasseAbstraite(int c) {
        champ = c;
    }
}
```



Compilation & exécution :


```
[14:12][Prog pc666 :]$ javac ClasseConcrete.java
[14:12][Prog pc666 :]$ java ClasseConcrete
Enfin définie
```

```
class ClasseConcrete extends ClasseAbstraite {
    int autreChamp;

    ClasseConcrete(int a, int c) {
        super(c);
        autreChamp = a;
    }

    void methodeAbstraite() {
        System.out.println("Enfin définie");
    }

    public static void main(String args[]) {
        ClasseConcrete cc = new ClasseConcrete(10, 20);
        cc.methodeAbstraite();
    }
}
```



Remarques

Si une classe :

- ▶ hérite d'une classe abstraite,
- ▶ et qu'elle ne définit pas toutes les méthodes abstraites,

alors cette classe sera aussi déclarée avec `abstract`.

Classe abstraite : utilité 4/4

Les classes/méthodes abstraites permettent de :


- ① structurer un arbre d'héritage (servant généralement de racine) ;
- ② imposer un cadre de programmation pour la conception d'un projet en :
 - consacrant des noms particuliers de méthodes (abstraites) ;
 - définissant des champs/méthodes communs pour l'ensemble de l'arbre d'héritage.
- ③ renforcer le polymorphisme par référencement avec un type abstrait.

```
abstract class Compte {
    static int cptCompte;
    int idCpt;
    int solde;

    Compte(int s) {
        idCpt = cptCompte++;
        solde = s;
    }

    abstract boolean estRentable();

    public String toString() {
        return "ID :" + idCpt +
            " solde : " + solde;
    }
}
```



```
class ComptePersonne extends Compte {
    String nomTitulaire;

    ComptePersonne(int s, String n) {
        super(s);
        nomTitulaire = n;
    }

    boolean estRentable() {
        return (solde > 10000);
    }

    public String toString() {...}
}
```

```
class CompteEntreprise extends Compte {
    String nomEntreprise;
    int capitalisation;

    CompteEntreprise(int s, String ne, int c) {
        super(s);
        nomEntreprise = ne;
        capitalisation = c;
    }

    boolean estRentable() {
        return (solde > (capitalisation / 10));
    }

    public String toString() {...}
}
```


Interface – définition 1/4

Une interface est un ensemble constitué :

- ❶ de constantes ;
- ❷ de méthodes abstraites ;
- ❸ des méthodes « par défaut » (depuis JAVA 8).

► Syntaxe :

```
interface NomInterface {
    [type nomChamp1;
    ...]
    [type nomMethode1;
    ...]
    [default type nomMethodeDefaut1 {definition}
    ...]
}
```

► Propriétés implicites des champs :

- final et public.

► Propriétés implicites des méthodes :

- public.

⇒ Préfixer par public les définitions effectives...

interface vs. abstract class

Interface et classe abstraite sont semblables. Toutefois, une classe abstraite peut contenir :

- des définitions de méthodes ;
- des attributs non constants.

Interface – usage 2/4

En POO, une classe ne peut hériter que d'au plus une seule classe.

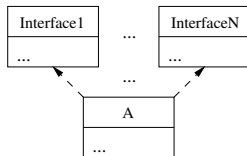
C'est une limitation forte !

En revanche, une classe peut hériter d'un nombre arbitraire d'interfaces.

Permet de moduler « l'interface » d'une classe ✓

La syntaxe JAVA et la représentation UML seront :

```
class A implements Interface1,..., InterfaceN {
    ...
}
```



Le programmeur :

- ▶ disposera des constantes fournies par ces interfaces ;
- ▶ devra définir toutes les méthodes de ces interfaces (ou sinon déclarer cette classe comme `abstract`).

Interface – en pratique 3/4

Plus encore que les classes abstraites, dans un projet, une interface permet :

- 1 consacrer des noms de méthodes ;
- 2 de moduler les capacités d'interaction d'une classe ;
- 3 structurer un projet logiciel.

En pratique, et notamment dans l'API, le nom d'une interface renseigne sur une certaine capacité dont devra jouir la classe l'implémentant.

Nom terminant par « *-able* »

Mis en *italique* dans l'API

Exemple : l'interface *Cloneable* de la classe `Object`.

Enfin, les interfaces :

- ▶ définissent des types (renforçant le polymorphisme par référencement d'objet) ;
- ▶ peuvent très bien hériter de plusieurs interfaces.

```
interface A extends Interface1, ..., InterfaceN {  
    ...  
}
```

Interface – exemple 4/4

```
interface Remboursable {
    int rembourser();
}

interface Echangeable {
    int echanger();
}

class BilletTGV {
    String nomClient;
    int prixBillet;

    BilletTGV(String n, int p) {
        nomClient = n;
        prixBillet = p;
    }
}
```

```
class BilletOuiGo
    extends BilletTGV
    implements Echangeable {
    boolean avecBagageSupp;

    BilletOuiGo(String n, int p,
        boolean abs) {
        super(n, p);
        avecBagageSupp = abs;
    }

    public int echanger() {
        int difference = 0;
        // Instructions d'échange.
        return difference;
    }
}
```

```
class BilletLoisir
    extends BilletTGV
    implements Remboursable,
        Echangeable {
    boolean avecPlaceCalme;
    boolean avecPrise;

    BilletLoisir(String n, int p,
        boolean apc, boolean ap) {
        super(n, p);
        avecPlaceCalme = apc;
        avecPrise = ap;
    }

    public int rembourser() {
        int somme = 0;
        // Instructions remboursement.
        return somme;
    }

    public int echanger() {
        int difference = 0;
        // Instructions d'échange.
        return difference;
    }
}
```

```
class SNCF {
    public static void main(String args[]) {
        BilletTGV b = new BilletOuiGo("Einstein", 40, true);
        System.out.println(b);
        Remboursable r = new BilletLoisir("Curie", 100, true, true);
        System.out.println(r);
    }
}
```



```
[18:42][Prog pc666 :]$ java SNCF
BilletOuiGo@6bc7c054
BilletLoisir@232204a1
```

Classe abstraite, interface & javac

Les classes abstraites et les interfaces sont :

- ① omniprésentes dans l'API ;
- ② très utilisées pour encadrer la programmation d'un projet.

Le débutant peine souvent avec javac pour compiler :

```


1 interface Assurable {
2     int rembourserVol();
3     int rembourserAnnulation();
4     int rembourserRetard();
5 }
6
7
8 class BilletTGV implements Assurable {
9     String nomClient;
10    int prixBillet;
11    ...

```

```

12    BilletTGV(String n, int p) {
13        nomClient = n;
14        prixBillet = p;
15    }
16
17    int rembourserRetard() {
18        int somme = 0;
19        // Instructions pour remboursement si retard.
20        return somme;
21    }
22 }

```



En effet, le compilateur javac veille au respect des concepts :

```

[09:58][Prog pc666 :]$ javac BilletTrainAssure.java
BilletTrainAssure.java:8: error: BilletTGV is not abstract and does not override abstract method
    rembourserAnnulation() in Assurable
class BilletTGV implements Assurable {
~
BilletTrainAssure.java:17: error: rembourserRetard() in BilletTGV cannot implement rembourserRetard() in Assurable
    int rembourserRetard() {
    ~
    attempting to assign weaker access privileges; was public
2 errors

```

1 Exceptions

- Intérêt
- Bloc try/catch
- Bloc finally

2 Classe Abstraite – Interface

- Classe Abstraite
- Interface

3 Duplication d'objets

- Surface/Profondeur
- Clonage d'objet

4 Quelques approfondissements

- La classe Object et ses méthodes
- Introspection en JAVA
- Classes génériques
- Classes enveloppes
- Collections
- Types énumérés

La copie d'objets – comment dupliquer un objet ?

Soit une classe Identite composée d'un champ **Biometrie** :

```
class Biometrie {
    int tailleCM;
    String couleurYeux;
    String couleurCheveux;

    Biometrie(int t, String cy, String cc) {
        ... // i.e. affectations ordinaires.
    }

    public String toString() {
        return ", taille : " + tailleCM +
            ", yeux : " + couleurYeux +
            ", cheveux : " + couleurCheveux;
    }
}
```

```
class Identite {
    String nom;
    int age;
    Biometrie infoBiometrique;

    Identite(String n, int a, Biometrie ib) {
        ... // i.e. affectations ordinaires.
        infoBiometrique = ib;
    }

    public String toString() {
        return "Nom : " + nom + ", age : " + age +
            infoBiometrique;
    }

    Identite copieIdentite() {
        return new Identite(this.nom, this.age,
            this.infoBiometrique);
    }
}
```

Méthode de duplication

```
class Copie {
    public static void main(String args[]) {
        Identite id1 = new Identite("Kahlo", 47, new Biometrie(159, "Marron", "Brun"));
        System.out.println(id1);
        Identite copieId1 = id1.copieIdentite();
        System.out.println("Copie id1 : " + copieId1);
        System.out.println("Les champs infoBiometrique sont-ils distincts ?");
        System.out.println(id1.infoBiometrique != copieId1.infoBiometrique);
    }
}
```

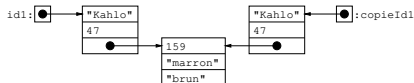


La copie d'objets – Surface vs. Profondeur

À l'exécution, nous avons :

```
[19:30][Prog pc666 :]$ java Copie
Nom : Kahlo, age : 47, taille : 159, yeux : Marron, cheveux : Brun
Copie de id1 : Nom : Kahlo, age : 47, taille : 159, yeux : Marron, cheveux : Brun
Les champs infoBiometrique sont-ils distincts ?
false
```

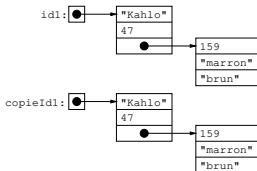
Schématiquement, à l'issue de la « copie », la situation est la suivante :



Seule la référence du champ
infoBiometrique a été
dupliquée!

⇒ Copie en surface

Alors que nous souhaitons :



L'objet de la référence du
champ infoBiometrique a
aussi été dupliqué...

⇒ Copie en profondeur

La copie d'objets en profondeur – méthode clone()

Il faudrait aussi disposer d'une méthode de duplication dans la classe Biometrie.

```
class Biometrie {
    ...
    Biometrie copieBiometrie() {
        return new Biometrie(tailleCM,
                             couleurYeux,
                             couleurCheveux);
    }
}
```

```
class Identite {
    ...
    Identite copieIdentite() {
        return new Identite(nom,
                             age,
                             infoBiometrique.copieBiometrie());
    }
}
```

À l'instar de la méthode toString(), la classe Object contient la méthode :

```
protected Object clone() throws CloneNotSupportedException
```

- ▶ Cette méthode effectue une copie en surface d'un objet.
- ▶ Pour l'utiliser,
 - implémenter l'interface *Cloneable*¹,
 - généralement, augmenter l'accessibilité (protected → public).

```
class Identite implements Cloneable {
    ...
    public Object clone() throws CloneNotSupportedException {...}
}
```

¹ Cette interface est... vide ! Étrangeté dans JAVA.

La copie d'objet – le cas des tableaux

La classe `System` dispose de la méthode statique `arraycopy()` :

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos,
                             int length) throws IndexOutOfBoundsException,
                                             ArrayStoreException,
                                             NullPointerException
```

Un tableau implémente aussi la méthode `clone()` :

```
[09:46][Prog pc666 :]$ java CopieSurfaceTableau
1 2 3 4
1 2 3 4
```

⇒ cast inutile pour la valeur de retour de `clone()` (cf. API)

```
class CopieSurfaceTableau {
    public static void main(String args[]) {
        int tabEntier[] = {1, 2, 3, 4};
        int copieTab[] = new int[tabEntier.length];
        // Copie avec arraycopy :
        System.arraycopy(tabEntier, 0, copieTab, 0,
                         tabEntier.length);

        // Copie avec clone() :
        int cloneTab[] = tabEntier.clone();
        // Affichage de la copie et du clone :
        for (int elt : copieTab) {
            System.out.print(elt + " ");
        }
        System.out.println();
        for (int elt : cloneTab) {
            System.out.print(elt + " ");
        }
        System.out.println();
    }
}
```



La copie d'objets en profondeur – méthode clone()

Copie en profondeur : copier récursivement et implémenter l'interface *Cloneable* dans les classes appropriées.

```
class Biometrie implements Cloneable {
    ...
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Identite implements Cloneable {
    ...
    public Object clone() throws CloneNotSupportedException {
        Identite copie = (Identite) super.clone();
        copie.infoBiometrique = (Biometrie) infoBiometrique.clone();
        return copie;
    }
}

class CopieAvecCloneProfondeur implements Cloneable {
    public static void main(String args[]) throws CloneNotSupportedException {
        Identite id1 = new Identite("Kahlo", 47, new Biometrie(159, "Marron", "Brun"));
        Identite copieId1 = (Identite) id1.clone();
        System.out.println("Les champs infoBiometrique sont-ils distincts ?");
        System.out.println(id1.infoBiometrique != copieId1.infoBiometrique);
    }
}
```

⇒ Bien caster les valeurs de retour de clone()

1 Exceptions

- Intérêt
- Bloc try/catch
- Bloc finally

2 Classe Abstraite – Interface

- Classe Abstraite
- Interface

3 Duplication d'objets

- Surface/Profondeur
- Clonage d'objet

4 Quelques approfondissements

- La classe Object et ses méthodes
- Introspection en JAVA
- Classes génériques
- Classes enveloppes
- Collections
- Types énumérés

Le paquetage `java.lang` – aperçu

D'après l'API `JAVA`, ce paquetage contient notamment :

- ▶ la classe `Object` ;
- ▶ les classes `String` et `StringBuffer` ;
- ▶ les classes « enveloppes » ;
- ▶ la classe `Math` ;
- ▶ la classe `System` ;
- ▶ la classe `Enum` ;
- ▶ la classe `Class` ;
- ▶ la classe `Thread` ;
- ▶ les classes `Throwable`, `Exception` et `Error` ;
- ▶ les interfaces `Cloneable`, `Comparable`, `Runnable`.

⇒ Réaliser des E/S

⇒ Exploiter les coeurs de votre CPU

Rappel sur le paquetage `java.lang`

Ce paquetage est automatiquement traité par le compilateur pour tous vos programmes.

⇒ inutile de mettre l'instruction `import` !

La classe Object – ses méthodes

Classe fondamentale : elle constitue la racine de l'arbre d'héritage. De plus, elle propose notamment les méthodes :

- ▶ `clone()` ;
⇒ permet de dupliquer des objets.
- ▶ `equals()` ;
⇒ permet de comparer des objets.
- ▶ `finalize()` ;
⇒ permet de définir un traitement à exécuter lors de l'application du ramasse-miette sur cet objet.
- ▶ `getClass()` ;
⇒ permet de connaître la classe propre de l'objet.
- ▶ `toString()`.

Hormis `getClass()`, le programmeur redéfinira généralement ces méthodes.

La méthode equals()

- ▶ Par défaut la méthode equals() de prototype

```
public boolean equals(Object obj)
```

permet de tester si deux variables indiquent le même objet en mémoire.

```
a.equals(b);    ⇔    (a == b)
```

- ▶ Redéfinir equals() pour tester l'égalité en contenu de deux objets.

```
import static java.lang.System.out;

class Etudiant {
    ... // Deux attributs age et nom + constructeur.
    public boolean equals(Object obj) {
        return ( (obj.getClass() == Etudiant.class) &&
                (((Etudiant) obj).age == age) &&
                (nom.compareTo(((Etudiant) obj).nom) == 0) );
    }

    public static void main(String args[]) {
        String message = "Un nobel et une oubliée...";
        Etudiant e1 = new Etudiant("Crick", 66);
        Etudiant e2 = new Etudiant("Franklin", 38);
        Etudiant eCopie = new Etudiant("Franklin", 38);
        out.println("e1 = message ? " + e1.equals(message));
        out.println("e1 = e2 ? " + e1.equals(e2));
        out.println("e2 = eCopie ? " + e2.equals(eCopie));
    }
}
```

● Abrège l'écriture des println().

● Comparaison des types avant accès aux champs.

● Comparaison lexicographique de deux chaînes de caractères.

À l'exécution :

```
[17:24][Prog pc666 :]$ java Etudiant
e1 = message ? false
e1 = e2 ? false
e2 = eCopie ? true
```

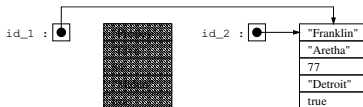
La méthode finalize()¹

- ❶ Les objets disposent d'un espace en mémoire tant qu'ils sont référencés par le programme.
- ❷ Si un objet n'est plus référencé, il est candidat au ramasse-miettes.

► Supposons que :

```
...
public static void main(String args[]) {
    Identite id_1, id_2;
    id_1 = new Identite("Presley", "Elvis",
                        84, "Memphis", true);
    id_2 = new Identite("Franklin", "Aretha",
                        77, "Detroit", true);
    ...
    System.out.println(id_1.estValide());
    id_1 = id_2;
    ...
}
// Transparent 60 du CM1.
```

► Après l'instruction surlignée, nous aurions en mémoire :



L'objet initialement référencé par `id_1` peut être détruit.

- ❸ Un objet soumis au ramasse-miettes invoquera sa méthode `finalize()` spécialement redéfinie :

```
protected void finalize() {
    // Traitements à effectuer
}
```

⇒ sauvegarde, libération de ressources « systèmes ».

¹ Depuis JAVA 9, cette méthode est cependant obsolète (i.e. marquée *deprecated*).

La méthode getClass() – introduction à l'introspection

L'introspection¹ est la capacité d'un programme à s'examiner dynamiquement.

Cela permet de connaître la structuration de la classe :

- ▶ les noms des champs, leurs types, leurs modificateurs d'accès ;
- ▶ les noms des méthodes, leurs signatures, leurs accessibilités, les exceptions associées ;

Le JDK propose l'outil javap permettant au programmeur d'examiner les classes.

```
class Etudiant {  
    String nom;  
    String prenom;  
    int age;  
  
    Etudiant(String n, String p, int a) {  
        nom = n;  
        prenom = p;  
        age = a;  
    }  
  
    public String toString() {  
        return "Nom : " + nom +  
               "\nPrenom : " + prenom +  
               "\nAge : " + age;  
    }  
}
```

Exemple :

```
[08:59][Prog pc666 :]$ javap Etudiant.class  
Compiled from "Introspection.java"  
class Etudiant {  
    java.lang.String nom;  
    java.lang.String prenom;  
    int age;  
    Etudiant(java.lang.String, java.lang.String, int);  
    public java.lang.String toString();  
}
```

¹ On parle aussi de réflexivité.

La méthode getClass() – introspection par la JVM

Toute classe chargée en mémoire dispose d'un champ statique référençant un objet de type Class.

Cet objet statique de type Class<?> peut s'obtenir de deux manières :

```
Class<?> obj = NomDeClasse.class;
```

```
Class<?> obj = objQuelconque.getClass();
```

Exemple :

```
// Au sein de la classe Etudiant
public static void main(String args[]) {
    Class<?> objClass = Etudiant.class;
    System.out.println("Affichage 1 : " + objClass);
    Etudiant e = new Etudiant("Franklin",
                              "Rosalind",
                              38);
    Class<?> objClassPourE = e.getClass();
    System.out.println("Affichage 2 : " + objClassPourE);
    System.out.println("Affichage 3 : " + objClassPourE.equals(objClass));
}
```



```
[09:38] [Prog pc666 :]$ java Etudiant
Affichage 1 : class Etudiant
Affichage 2 : class Etudiant
Affichage 3 : true
```

La méthode getClass() – l'accès à la structure de l'objet 1/2

Un tel objet permet d'examiner pour ce type :

- ▶ son nom,

```
String nomDeClasse = objClass.getName();  
// Sa fonction réciproque :  
autreObjClass = Class.forName("NomDeClasse");
```

- ▶ ses champs (avec le type Field),

```
// Obtention des champs déclarés d'une classe :  
Field [] champsObj = objClass.getDeclaredFields()
```

- ▶ ses méthodes (avec le type Method),

```
// Obtention des méthodes déclarées d'une classe :  
Method [] champsObj = objClass.getDeclaredMethods()
```

- ▶ ses constructeurs (avec le type Constructor<?>).

```
// Obtention des constructeurs déclarés d'une classe :  
Constructor<?> [] construteursObj = objClass.getDeclaredConstructors();
```

La méthode getClass() – l'accès à la structure de l'objet 2/2

Pour ces trois catégories Field/Methods/Constructor<?>, il est possible d'examiner :

- ▶ les types (avec le type Class<?> ou Class<?> []);

```
Class<?> typeDeChamp = objField.getType();
Class<?> typeDeRetour = objMethod.getReturnType();
Class<?> tabTypesDesParametres[] = objMethod.getParameters.Types();
... // et aussi les exceptions.
```

- ▶ les modificateurs d'accès des champs/méthodes (avec le type Modifier).

```
Modifier objModificateurAcces = objField.getModifier();
objModificateurAcces.isPublic();
objModificateurAcces.isPrivate();
objModificateurAcces.isProtected();
objModificateurAcces.isStatic();
objModificateurAcces.isFinal();
objModificateurAcces.isAbstract();
... // Cf. API JAVA.
```

L'introspection est pratique :

...mais aussi risquée :

⇒ création d'objet à la volée.

⇒ jouer avec l'encapsulation !

Exemple avec la classe Class

Ce programme permet d'obtenir l'ascendance d'une classe spécifiée sur la ligne de commandes en utilisant les méthodes :

- ▶ `getSuperclass()`
- ▶ `getName()`
- ▶ `forName()`

```
class Ascendance {
    static int afficherParent(Class<?> objClass) { // Méthode récursive.
        String branchement = "";
        String indentation = "";
        int distRacine = 0;
        if (objClass != Object.class) {
            branchement = "\\ ";
            indentation = " ";
            distRacine = 1 + afficherParent(objClass.getSuperclass());
        }
        // Création d'une indentation adaptée.
        for (int i = 1; i < distRacine; i++) {
            indentation += " ";
        }
        System.out.println(indentation + branchement + objClass.getName());
        return distRacine;
    }

    public static void main(String args[]) throws ClassNotFoundException,
        IllegalAccessException,
        InstantiationException {
        Class<?> objClass = Class.forName(args[0]);
        afficherParent(objClass);
    }
}
```

À l'exécution (en précisant bien le nom complet) :

```
[09:59][Prog pc666 :]$ java Ascendance java.lang.NumberFormatException
java.lang.Object
 \_java.lang.Throwable
   \_java.lang.Exception
     \_java.lang.RuntimeException
       \_java.lang.IllegalArgumentException
         \_java.lang.NumberFormatException
```

Une autre forme d'introspection : les annotations – aperçu

Une annotation est une information définie par le programmeur et accessible :

- ① au compilateur javac ;
- ② à la JVM ;
- ③ aux objets eux-mêmes.

Une annotation est une étiquette codifiée/structurée et affectée de méta-données.

► Définition syntaxe :

```
modificateurAcces @interface NomAnnotation {
    type nomChampsMetaDonnee ();
    ...
}
```

► Usage dans le code :

```
@NomAnnotation (nomChampMetaDonnee = val1,...)
classe/champ/méthode
```

JAVA définit le type Annotation et propose des annotations standards :

@Deprecated @Generated
 @Override @SuppressWarnings ⇒ Indications de compilation.

Et « cerise sur le gâteau » : possibilité d'annoter les annotations...

@Documented @Inherited @Repeatable ⇒ Contraintes sur les annotations.
 @Retention @Target On parle de méta-annotations.

Des types paramétrés – un bref aperçu de la généricité

Définition : en programmation, une fonction ou une structure de données est dite générique lorsqu'elle s'affranchit des types qu'elle manipule.

▶ Exemple :

- une implémentation de « *quick sort* » trie tout type de donnée ;
- une liste doublement chaînée organise tout type de donnée.

JAVA permet de définir des méthodes ou des classes génériques paramétrées par des types (ici noté T).

▶ Syntaxe classe générique :

```
class<T,...> {
    T champ; // champ d'instance de type T
    ...
    methode(T arg,...)
    ...
}
```

▶ Syntaxe méthode générique :

```
// Si U n'est pas paramètre d'une classe hôte:
[static] <U,...> [U] methode(U arg,...)
// Si la classe hôte n'est pas générique :
[static] <T,...> [T] methode(T arg,...)
```

L'API JAVA et la généricité

- ▶ Les types paramètres peuvent être contraints ou sans limite (*joker*) : `<T extends Number>` et `<?>`.
- ▶ De très nombreuses classes utilisent la généricité : `Class<?>`, `ArrayList<?>`... **mais les tableaux génériques n'existent pas !**

Des types et des types

Rappel : un type en JAVA est soit « primitif » soit « référence » :

- ▶ les types primitifs,

boolean, byte, char, double, float, int, long, short

- ▶ les types références,

Object, String, StringBuffer...

Différences :

- 1 l'opération d'affectation ;

copie de la valeur vs. copie d'une adresse

- 2 règles de conversions distinctes ;

hiérarchie vs. arborescence

- 3 quid du polymorphisme (et donc d'usage de l'API) ?

pas de polymorphisme pour les types primitifs !

Problème : manque d'uniformité au sein des types.

Les classes « enveloppes » (UK/US : *wrappers*)

Pour chaque type primitif, il existe une classe enveloppe :

Boolean, Byte, Character, Double, Float, Integer, Long, Short

Exemple (avec ancienne/nouvelle syntaxe) :

```
// Construction et initialisation d'un objet Integer
Integer effectifEI2I4 = new Integer(30);
// Accès à la valeur contenue
System.out.println("Effectif EI2I4 : " + effectifEI2I4.intValue());

// Création et initialisation d'un objet Float
Float moyenneJava = 11.35f;
// Accès (et modification) de la valeur contenue
System.out.println("Moyenne Java : " + moyenneJava++);
```

Boxing/Unboxing manuel
(ancienne syntaxe).

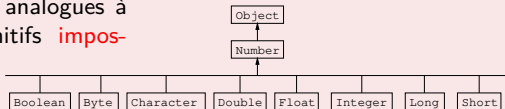
Boxing/Unboxing automatique

Attention : conversion/comparaison de *wrappers*

- ▶ Conversions implicites analogues à celles des types primitifs **impossibles**.

- ▶ Comparaison :

- == compare les adresses et non les contenus ;
- utiliser la méthode equals() (bien redéfinie par l'API).



Types enveloppes et API JAVA

L'API JAVA est très riche, notamment en structures de données :

- ▶ implémentant l'interface *Collection<E>* ;
- ▶ utilisant le cas échéant des objets de type *Iterator<E>* et *Comparator<E>*.

Pour manipuler un tableau d'entiers, JAVA utilisera des objets Integer.

Ici, nous avons :

- ▶ autoboxing
- ▶ utilisation d'une méthode de la classe Collections

```
import java.util.*;

class CollectionInteger {
    public static void main(String args[]) {
        ArrayList<Integer> tabEntier = new ArrayList<Integer>();
        for (int i = 0; i < 5; i++) {
            tabEntier.add((int) (Math.random() * 100));
        }
        for (int elt : tabEntier) {
            System.out.print(elt + " ");
        }
        System.out.println();
        System.out.println("Mélangeons le tout");
        Collections.shuffle(tabEntier);
        Iterator<Integer> iter = tabEntier.iterator();
        while (iter.hasNext()) {
            System.out.print(iter.next() + " ");
        }
        System.out.println("");
    }
}
```

À l'exécution.

```
[16:25][Prog pc666 :]$ java CollectionInteger
6 97 14 40 31
Mélangeons le tout
31 97 6 40 14
```

La classe Collections – boîte à outils algorithmiques

Comme la classe Math, la classe Collections¹ est composée uniquement de méthodes de classe opérant sur :

- ▶ les tableaux,
- ▶ les structures de données de l'API (*i.e.* *Collection*, *List*...).

On trouvera notamment des méthodes telles que :

```
static boolean disjoint(Collection<?> c1, Collection<?> c2)
static int frequency(Collection<?> c, Object o)
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)
static void reverse(List<?> list)
static void shuffle(List<?> list)
static <T> void sort(List<T> list, Comparator<? super T> c)
static void swap(List<?> list, int i, int j)
...
```

JAVA et ses implémentations

Attention : l'API précise que le comportement de certaines de ces méthodes dépend de l'implémentation.

¹<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

Les types énumérés en JAVA – le type Enum

Rappel de programmation : les énumérations

Dans un langage, une énumération est une commodité d'écriture permettant d'associer des noms à des valeurs constantes (généralement des entiers).

JAVA considère les énumérations comme des classes, héritant du type Enum :

► Syntaxe :

```
enum NomEnum {id1,..., idN}
```

≠ C : ce ne sont pas des entiers !

► Accès/Affectation :

```
NomEnum varEnum;  
varEnum = NomEnum.id1;
```

✓ : définit un type !

► Syntaxe façon « classe » :

```
enum NomEnum {  
    id1 [(init1)],..., idN [(initN)];  
    Champs  
    Constructeurs  
    Methodes  
}
```

► Principales méthodes :

TypeEnum.idX.toString();	→ "idX"
NomEnum.valueOf("idX");	→ idX
varEnum.ordinal();	→ rang(idX)
varEnum1.compareTo(varEnum2);	→ rang(varEnum1) - rang(varEnum2)
TypeEnum.values();	→ NomEnum tab[]

Les types énumérés en JAVA - exemple

```
enum Mois {JANVIER, FEVRIER, MARS,
           AVRIL, MAI, JUIN,
           JUILLET, AOUT, SEPTEMBRE,
           OCTOBRE, NOVEMBRE, DECEMBRE
}

class Bulletin {
    Mois mois;
    String nom;

    Bulletin(Mois m, String n) {
        mois = m;
        nom = n;
    }

    int trimestre() {
        int numTrimestre = 0;
        switch (this.mois) {
            case JANVIER: case FEVRIER: case MARS:
                numTrimestre = 1;
                break;
            case AVRIL: case MAI: case JUIN:
                numTrimestre = 2;
                break;
            case JUILLET: case AOUT: case SEPTEMBRE:
                numTrimestre = 3;
                break;
            case OCTOBRE: case NOVEMBRE: case DECEMBRE:
                numTrimestre = 4;
                break;
            default : numTrimestre = -1;
        }
        return numTrimestre;
    }
}
```

```
class ExempleEnum {
    public static void main(String args[]) {
        Bulletin b1 = new Bulletin(Mois.MAI, "Meitner");
        System.out.println(b1.trimestre());

        Bulletin b2 = new Bulletin(Mois.OCTOBRE, "Mirzakhani");
        System.out.println(b2.trimestre());
        System.out.println(b1.mois.compareTo(b2.mois));

        Bulletin b3 = new Bulletin(Mois.OCTOBRE, "Joliot-Curie");
        System.out.println(b2.mois == b3.mois);

        Mois tabMois[] = Mois.values();
        for (int i = 0; i < tabMois.length; i++) {
            System.out.print(tabMois[i] + " ");
            if ((i + 1) % 3 == 0) {
                System.out.println();
            }
        }
    }
}
```



À l'exécution :

```
[17:27][Prog pc666 :]$ java ExempleEnum
2
4
-5
true
JANVIER FEVRIER MARS
AVRIL MAI JUIN
JUILLET AOUT SEPTEMBRE
OCTOBRE NOVEMBRE DECEMBRE
```

Retour sur les types en JAVA (1/2)

Au final, tout type est soit :

① primitif

- char
- byte, short, int, long
- float, double
- boolean

② référence

- Tableaux
- Classes de l'API
- Classes du programmeur
- les classes abstraites
- les interfaces
- Enum
- Annotation

avec conversions internes au niveau de ces familles selon :

une hiérarchie des types

l'arborescence d'héritage

JAVA, les pointeurs et la mémoire

- ▶ Pas de pointeur en JAVA, mais des « références » aux objets ;
- ▶ La référence null est à éviter ;
- ▶ Pas d'arithmétique des pointeurs (≠ C/C++);
- ▶ Pas de gestion manuelle de la mémoire...malgré la méthode de nettoyage :

```
public void static gc() // (cf. API, classe System)
```

Retour sur les types en JAVA (2/2)

Rappel : le compilateur déduit le type des expressions mixtes ¹.

Exemple :

```
int i = 12;  
double d = 3.14;  
d + i;
```

javac déduit que cette expression est de type double.

Avec JAVA 10, le mot clé « var » permet de déclarer des identifiants dont le type sera déterminé à la compilation.

Exemple :

```
int i = 12;  
double d = 3.14;  
var resultat = d + i;
```

javac déduit que resultat sera de type double.

Déclaration typée « var » – intérêts & écueil

- + évite quelques redondances (facilité d'écriture)
- + facilite la vie du programmeur avec les types complexes/génériques
- point trop n'en faut... sinon le code devient incompréhensible !

¹Revoir le CM1.